Model Checking Languages of Data Words*

Benedikt Bollig[1], Aiswarya Cyriac[1], Paul Gastin[1], and K. Narayan Kumar[2]

[1] LSV, ENS Cachan, CNRS & INRIA, France
{bollig,cyriac,gastin}@lsv.ens-cachan.fr
[2] Chennai Mathematical Institute, India
kumar@cmi.ac.in

Abstract. We consider the model-checking problem for data multi-pushdown automata (DMPA). DMPA generate data words, i.e, strings enriched with values from an infinite domain. The latter can be used to represent an unbounded number of process identifiers so that DMPA are suitable to model concurrent programs with dynamic process creation. To specify properties of data words, we use monadic second-order (MSO) logic, which comes with a predicate to test two word positions for data equality. While satisfiability for MSO logic is undecidable (even for weaker fragments such as first-order logic), our main result states that one can decide if all words generated by a DMPA satisfy a given formula from the full MSO logic.

1 Introduction

In recent years, there has been an increasing interest in data words and data trees, i.e., structures over an infinite alphabet. Data trees may serve as a model of XML documents where the data part refers to attribute values or text contents [3]. Data words, on the other hand, are suitable to model the behavior of concurrent programs where an unbounded number of processes communicate via message passing [4, 5].

Naturally, a variety of formalisms have been considered to specify sets of data words in the context of verification. A considerable amount of work has gone into the study of temporal and monadic second-order (MSO) logic, mainly focusing on the satisfiability problem [2, 8, 9, 19]. MSO logic over data words allows us to check data values of two word positions for equality. However, the logic is so complex that only severely restricted fragments preserve its decidability. A remarkable result due to Bojańczyk et al. states that satisfiability is decidable for first-order logic when it is restricted to two variables [2], albeit of very high complexity, as it is equivalent to reachability for Petri nets. Elementary upper bounds were only obtained by restricting the logic further [8,19]. Anyway, decidability crucially relies on the fact that there is only one data value per position, which is clearly not sufficient to model executions of concurrent message-passing programs. Indeed, the lack of expressiveness and extensibility of those logics limits their use for verification.

* Supported by LIA InForMel, ARCUS, and DIGITEO LoCoReP.

In this paper, we consider the model-checking problem, which has not received as much attention in the context of data words as satisfiability, and we adopt the orthogonal approach of restricting the domain of data words instead of pruning the logic. More precisely, we introduce data multi-pushdown automata (DMPA), which may, for example, represent the behavior of a concurrent program. Requirements specifications for such languages can then be written in the full MSO logic. Our main result is that the model-checking problem is decidable: Do all data words accepted by a DMPA satisfy a given MSO formula?

Like automata over finite alphabets, a DMPA uses standard building blocks such as states and stack symbols. Moreover, it has (finitely many) registers, which can store concrete data values in a run. Unlike a simple pushdown automaton, a DMPA is equipped with several stacks and can define non-context free behaviors. However, while multi-pushdown automata over a finite alphabet are often used for the verification of concurrent recursive programs [15, 16], modeling recursion is not our primary goal. Rather, in the context of an unbounded number of processes, context-sensitive rewriting is necessary to describe distributed protocols, as they typically operate in several phases. For example, DMPA are able to model a token-based leader election protocol where the number of processes is unknown. Though such a protocol can be implemented locally in terms of finite-state processes, their global, observable behavior is not context-free.

Our decidability proof relies on the following idea: A tree-like structure in terms of (multiply) nested words over a *finite* alphabet is built on top of a data word and is used to recover word positions that carry the same data value. Nested words naturally appear as runs of DMPA. To preserve decidability of MSO model checking, as we deal with several stacks, we have to impose a bound on the number of switches from one stack to another. Model checking DMPA can then be reduced to satisfiability of MSO logic over nested words with a bounded number of phases, which is decidable due to [15].

At first glance, DMPA produce data words, which are linearly ordered and of course suitable to describe sequential behaviors. One important aspect of MSO logic, however, is that it can easily define causal dependencies between events that go beyond the linear order induced by a data word. Our approach is, therefore, not restricted to sequential systems, but allows us to model complex dynamic concurrent programs and protocols from mobile computing. Our hope is that this will help to bring data words closer to applications in verification.

Related work. A wide range of automata over data words have been introduced in the literature [2, 6, 12–14, 20]. For all of them, MSO model checking is undecidable. Moreover, none of them is suited to represent distributed protocols: they either run on one-dimensional data words, process a data word in several passes, or do not support concurrency. An automaton model that captures the interplay of communicating processes is due to [4]. Its modeling power, however, comes at the price of an undecidable emptiness problem.

Model checking of counter machines against freeze LTL was considered in [10,11]. That setting is quite different from ours, as formulas are interpreted over runs, which contain counter evaluations as data values. Moreover, the temporal

logic, which can be embedded into MSO logic, has to be restricted further to obtain decidability results.

Our approach of introducing DMPA as a model of programs and using unrestricted MSO as requirements-specification language is partly inspired by [17]. There, Leucker et al. consider dynamic message sequence graphs as a model of dynamic communicating systems where an unbounded number of processes communicate through message exchange. No link with data words was established, though, and rules are context-free so that a leader election protocol cannot be described. Thus, we provide a more general, but conceptually simple, framework with a generic proof of decidability of MSO model checking.

Outline. In Section 2, we introduce data words and DMPA. Section 3 presents MSO logic to specify properties over data words. In Section 4, we establish decidability of the model-checking problem. We conclude in Section 5.

2 Data Words and Data Multi-Pushdown Automata

By $\mathbb{N} = \{0, 1, 2, \ldots\}$, we denote the set of natural numbers. For $n \in \mathbb{N}$, we let $[n]$ denote the set $\{1, \ldots, n\}$. A *ranked alphabet* is a non-empty set Σ where every letter $a \in \Sigma$ has an arity, denoted $\mathit{arity}_\Sigma(a) \in \mathbb{N}$. We sometimes write $\mathit{arity}(a)$ instead of $\mathit{arity}_\Sigma(a)$ when Σ is clear from the context. For any set D, we let $\Sigma_D = \{a(d_1, \ldots, d_m) \mid a \in \Sigma,\ m = \mathit{arity}(a), \text{ and } d_1, \ldots, d_m \in D\}$.

Henceforth, we fix a finite ranked alphabet Σ (of *labels*) and an infinite set D (of *data values*). The elements of Σ_D are called *actions*. A *data word* is a sequence of actions, i.e., an element from Σ_D^*. Given a data word $w = w_1 \ldots w_n$ of length n, we denote by $\mathrm{dom}(w)$ its domain $\{1, \ldots, n\}$, i.e., its set of positions. For $i \in \mathrm{dom}(w)$ with $w_i = a(d_1, \ldots, d_m)$, we let $\mathit{label}(i)$ refer to a and $\mathit{data}_k(i)$ to d_k, for all $k \in \{1, \ldots, m\}$. Moreover, we set $\mathit{arity}(i) = m$. For example, if $D = \mathbb{N}$ and $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ with $\mathit{arity}(\mathsf{a}) = 1$ and $\mathit{arity}(\mathsf{b}) = 2$, then $\mathsf{a}(4)\,\mathsf{b}(7,9)\,\mathsf{a}(6)\,\mathsf{b}(10,7)$ is a data word from Σ_D^*. We have $\mathit{label}(3) = \mathsf{a}$ and $\mathit{data}_2(4) = 7$.

To represent systems whose executions are data words, we use data multi-pushdown automata (DMPA). Basically, a DMPA is a multi-pushdown automaton with $\mathbb{h} \geq 1$ stacks over some finite alphabet. In addition, it has $\mathbb{k} \geq 0$ global registers, $R = \{r_1, \ldots, r_\mathbb{k}\}$, which can store data values. Data values can also be stored on stacks along with a stack symbol from a finite ranked alphabet $\mathcal{Z}$. To refer to these data values, we use *parameters* from the infinite supply $P = \{p_1, p_2, \ldots\}$. A transition of a DMPA depends on the current state of the automaton, the data values that are stored in the registers, and the top symbol of the stack chosen by the transition as well as its associated data values. The data values that are stored along with $A \in \mathcal{Z}$ can be accessed by means of the parameters $P_A = \{p_1, \ldots, p_{\mathit{arity}(A)}\} \subseteq P$. Now, a transition is controlled by a guard that allows us to compare data values stored in registers with data values from the target stack. A *guard* (wrt. A) is generated by the grammar $\Phi ::= \mathit{true} \mid \pi_1 = \pi_2 \mid \Phi \wedge \Phi \mid \neg\Phi$ where $\pi_1, \pi_2 \in R \cup P_A$. For example, guard $r_1 = r_2 \wedge \neg(r_1 = p_3)$ requires that the contents of register r_1 equals the data

value held in r_2, but is different from the third data value stored on top of the target stack. If the guard is satisfied, the automaton outputs one or several actions that may use the data values represented by $R \cup P_A$. They may also use *fresh* data values, and we will use the parameters $Q = \{q_1, q_2, \ldots\}$ as placeholders for them. Finally, the transition updates the current state, the register contents, and the stacks. Register and stack updates are allowed to use stored data values as well as fresh ones. More precisely, an *update* (wrt. A) is a tuple $\mathsf{upd} = (\pi_1, \ldots, \pi_{\mathbb{k}}, u_1, \ldots, u_{\mathbb{h}})$. For $i \in [\mathbb{k}]$, $\pi_i \in R \cup P_A \cup Q$ determines the new data value stored in r_i. For example, if $\pi_i = p_1$, then r_i obtains the first value stored on top of the the target stack; if $\pi_i = r_i$, then r_i is left unchanged; if $\pi_i = q_j$, then r_i will get some fresh data value. For $t \in [\mathbb{h}]$, u_t is a string over $\mathcal{Z}_{R \cup P_A \cup Q}$, which, instantiated with data values, is pushed onto stack t. Again, this string may use data values stored in registers (R) or the target stack (P_A), as well as fresh data values (Q). Let us formally define DMPA.

Definition 1 (data multi-pushdown automaton). *Let $\mathbb{k} \geq 0$ and $\mathbb{h} \geq 1$. A ($\mathbb{k}$-register, $\mathbb{h}$-stack) data multi-pushdown automaton (DMPA) over (Σ, D) is a 6-tuple $\mathcal{A} = (S, \mathcal{Z}, s_0, Z, F, \Delta)$ where S is a finite* set of states*, $\mathcal{Z}$ is a finite ranked alphabet of* stack symbols*, $s_0 \in S$ is the* initial state*, $Z \in \mathcal{Z}$ is the* start symbol *with arity$(Z) = 0$, and $F \subseteq S$ is the set of* final states*. Moreover, Δ is a finite set of* transitions*. A transition δ is of the form*

$$t{:}A, s \xrightarrow{\Phi, u, \mathsf{upd}} s'$$

where $s, s' \in S$ are states, $t \in [\mathbb{h}]$ is a stack, $A \in \mathcal{Z}$, Φ is a guard wrt. A, $u \in (\Sigma_{R \cup P_A \cup Q})^*$, and upd is an update wrt. A. We let $\Pi_\delta = R \cup P_A \cup Q_\delta$ with Q_δ the set of parameters from Q occurring in u or upd.

We let $Conf_{\mathcal{A}} := S \times D^{\mathbb{k}} \times 2^D \times (\mathcal{Z}_D^*)^{\mathbb{h}}$ denote the set of *configurations* of $\mathcal{A}$. Configuration $\gamma = [s, \mathbf{r}, U, w_1, \ldots, w_{\mathbb{h}}]$ with $\mathbf{r} = (d_1, \ldots, d_{\mathbb{k}})$ says that the current state is s, the content of register r_i is d_i, the data values from U have already been used, and the stack contents are $w_1, \ldots, w_{\mathbb{h}}$ where we assume that the topmost symbol is written last. Now, consider a transition $\delta = t{:}A, s \xrightarrow{\Phi, u, \mathsf{upd}} s'$ with $\mathsf{upd} = (\pi_1, \ldots, \pi_{\mathbb{k}}, u_1, \ldots, u_{\mathbb{h}})$, which we call a t-transition since it pops A from stack t. It is enabled at γ if $w_t = w_t' A(d_1', \ldots, d_m')$ and $\sigma \models \Phi$ where $\sigma : R \cup P_A \to D$ is the interpretation defined by $\sigma(r_i) = d_i$ for $i \in [\mathbb{k}]$ and $\sigma(p_j) = d_j'$ for $j \in [m]$. In this case, for any extension of σ to Π_δ (still denoted σ) assigning to parameters in Q_δ pairwise distinct fresh values from $D \setminus U$, there is a concrete transition $\gamma \overset{\sigma(u)}{\Longrightarrow}_{\sigma,\delta} \gamma'$ where $\sigma(u)$ is the data word obtained from u by replacing any parameter $\pi \in \Pi_\delta$ occuring in u by $\sigma(\pi) \in D$, and

$$\gamma' = [s', (\sigma(\pi_1), \ldots, \sigma(\pi_{\mathbb{k}})), U \cup \sigma(Q_\delta), w_1 \sigma(u_1), \ldots, w_t' \sigma(u_t), \ldots, w_{\mathbb{h}} \sigma(u_{\mathbb{h}})] \; .$$

A configuration of the form $[s_0, (d_1, \ldots, d_{\mathbb{k}}), \{d_1, \ldots, d_{\mathbb{k}}\}, Z, \varepsilon, \ldots, \varepsilon]$ with the data values $d_1, \ldots, d_{\mathbb{k}}$ pairwise distinct is called *initial*, and a configuration $[s, \mathbf{r}, U, w_1, \ldots, w_{\mathbb{h}}]$ such that $s \in F$ is called *final*. A *run* of $\mathcal{A}$ on $w \in \Sigma_D^*$

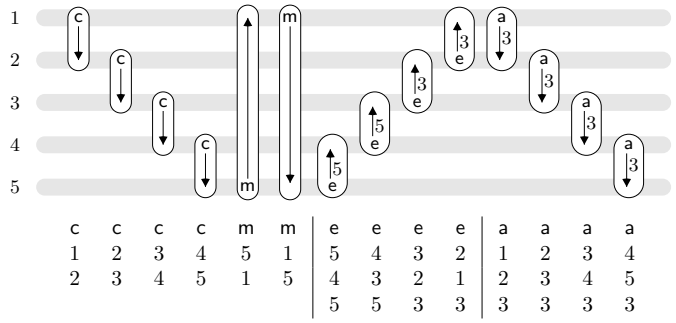

Fig. 1. A data word generated by the leader election protocol

is a sequence $\gamma_0 \overset{w_1}{\Longrightarrow}_{\sigma_1,\delta_1} \gamma_1 \overset{w_2}{\Longrightarrow}_{\sigma_2,\delta_2} \cdots \overset{w_n}{\Longrightarrow}_{\sigma_n,\delta_n} \gamma_n$ such that $w = w_1 \cdots w_n$ and γ_0 is initial. The run is *accepting* if γ_n is final. We let $L(\mathcal{A}) := \{w \in \Sigma_D^* \mid$ there is an accepting run of $\mathcal{A}$ on $w\}$ be the *language* of $\mathcal{A}$. Note that $L(\mathcal{A})$ is closed under permutation of data values.

It is easy to see that DMPA have an undecidable emptiness problem, even when we assume only two stacks as well as labels and stack symbols with arity 0. Therefore, we will restrict the number of *phases*, a notion that goes back to La Torre et al. who introduced it for multi-stack pushdown automata [15]. In one phase, one can only pop from one particular stack. Formally, for $\ell \geq 1$, a run $\gamma_0 \overset{w_1}{\Longrightarrow}_{\sigma_1,\delta_1} \gamma_1 \overset{w_2}{\Longrightarrow}_{\sigma_2,\delta_2} \cdots \overset{w_n}{\Longrightarrow}_{\sigma_n,\delta_n} \gamma_n$ is an *ℓ-phase run* if the sequence $\delta_1 \dots \delta_n$ can be split into ℓ blocks, each block using only t-transitions for some $t \in [\![h]\!]$. By $L_\ell(\mathcal{A})$, we then denote the restriction of $L(\mathcal{A})$ to data words that are accepted by ℓ-phase runs.

Remark 2. A DMPA that does not use its stacks (i.e., it never replaces the start symbol Z) corresponds to a restriction of fresh-register automata [20]. The restriction consists in allowing the automaton to read only data values that are either fresh or stored in the registers. In the terminology of [20], *local*-freshness transitions are discarded, while *global*-freshness transitions are permitted.

Example 3. We will specify a 1-register 2-stack DMPA that models the communication flow of a token-based leader election protocol. One possible behavior of the protocol is captured by the data word from Figure 1. The underlying alphabet of labels is $\Sigma = \{\mathsf{c}, \mathsf{m}, \mathsf{e}, \mathsf{a}\}$. The labels c and m have arity 2, and the labels e and a have arity 3. Data values from $D = \mathbb{N}$ will be used to model process identifiers (pids).

In the figure, a root process with pid 1 initiates a cascade of process spawns. It first executes action $\mathsf{c}(1, 2)$, which creates a new process with pid 2. Process 2 then executes $\mathsf{c}(2, 3)$ to create a new process 3, and so on. The number of processes created is not fixed apriori. The creation phase is followed by a message exchange between the very last process and the root, whereupon the former initiates the election phase. In the election phase, a process d non-deterministically chooses either the pid received from $d + 1$ or its own identity, and forwards it to

	s	stack	Φ	action	upd			s'
lep_1	s_0	$1{:}Z()$	*true*	ε	q_1	$Z()C(q_1)$	$Y()$	s_1
lep_2	s_1	$1{:}C(p_1)$	*true*	$\mathsf{c}(p_1, q_1)$	r_1	$X(p_1)E(q_1, p_1)C(q_1)$	ε	s_1
lep_3	s_1	$1{:}C(p_1)$	*true*	$\mathsf{m}(p_1, r_1)\mathsf{m}(r_1, p_1)$	p_1	ε	ε	s_2
lep_4	s_2	$1{:}E(p_1, p_2)$	*true*	$\mathsf{e}(p_1, p_2, r_1)$	r_1	ε	$A(p_2, p_1)$	s_2
lep_5	s_2	$1{:}X(p_1)$	*true*	ε	r_1	ε	ε	s_2
lep_6	s_2	$1{:}X(p_1)$	*true*	ε	p_1	ε	ε	s_2
lep_7	s_2	$1{:}Z()$	*true*	ε	r_1	ε	ε	s_3
lep_8	s_3	$2{:}A(p_1, p_2)$	*true*	$\mathsf{a}(p_1, p_2, r_1)$	r_1	ε	ε	s_3
lep_9	s_3	$2{:}Y()$	*true*	ε	r_1	ε	ε	s_4

Fig. 2. A DMPA for the leader election protocol

$d-1$ by executing $\mathsf{e}(d, d-1, \ell)$. In the following announcement phase, the pid ℓ of the elected leader is forwarded to all the processes, by executing actions of the form $\mathsf{a}(d, d+1, \ell)$. The figure depicted on top of the data word illustrates the creation, election, and announcement phases and the processes involved in their actions. Recall that data values have no meaning and may only be checked for equality, and we could have assumed any possible permutation of pids.

Figure 2 depicts a 1-register 2-stack DMPA $\mathcal{A}_{lep} = (S, \mathcal{Z}, s_0, Z, F, \Delta)$ for the leader election protocol. Hereby, $S = \{s_0, \ldots, s_4\}$, $F = \{s_4\}$, and $\mathcal{Z} = \{Z, Y, X, C, E, A\}$ where $arity(Z) = arity(Y) = 0$, $arity(X) = arity(C) = 1$, and $arity(E) = arity(A) = 2$. Moreover, Δ contains 9 transitions, $\mathsf{lep}_1, \ldots, \mathsf{lep}_9$.

A run of $\mathcal{A}_{lep}$ involving four processes is given in Figure 3 (we omit the renamings involved in transitions). The transitions $\mathsf{lep}_1, \mathsf{lep}_2, \mathsf{lep}_3$ put up the creation phase, represented by the upper part of the figure. For every action $\mathsf{c}(d, d+1)$ that is produced, $X(d)E(d+1, d)$ is written onto the first stack to be used in the election phase. Simultaneously, the topmost stack symbol $C(d+1)$ stores the process $d+1$ that has to perform the following action. During the creation phase, the register stores the identity 1 so that it can later execute $\mathsf{m}(4, 1)\mathsf{m}(1, 4)$. The election phase is performed by transitions lep_4 to lep_6. Here, the register stores the current leader ℓ which is sent to the next process in lep_4 with action $\mathsf{e}(d+1, d, \ell)$. Moreover, $A(d, d+1)$ is written onto the second stack to prepare the announcement phase. Then, process d chooses either to preserve the current leader in lep_5 or to switch the leader to itself in lep_6. Transition lep_7 triggers the announcement phase which is performed by lep_8 where the final leader ℓ stored in the register is sent to all processes with $\mathsf{a}(d, d+1, \ell)$. Note that, lep_8 causes the only control change, from the first to the second stack. We actually have $L(\mathcal{A}_{lep}) = L_2(\mathcal{A}_{lep})$. □

$$
\begin{array}{rllll}
 & [s_0 & 0 & \{0\} & Z() & \varepsilon] \\
\xrightarrow{\varepsilon}_{\mathsf{lep}_1} & [s_1 & 1 & \{0,1\} & Z()C(1) & Y()] \\
\xrightarrow{\mathsf{c}(1,2)}_{\mathsf{lep}_2} & [s_1 & 1 & \{0,1,2\} & Z()X(1)E(2,1)C(2) & Y()] \\
\xrightarrow{\mathsf{c}(2,3)}_{\mathsf{lep}_2} & [s_1 & 1 & \{0,\ldots,3\} & Z()X(1)E(2,1)X(2)E(3,2)C(3) & Y()] \\
\xrightarrow{\mathsf{c}(3,4)}_{\mathsf{lep}_2} & [s_1 & 1 & \{0,\ldots,4\} & Z()X(1)E(2,1)X(2)E(3,2)X(3)E(4,3)C(4) & Y()] \\
\xrightarrow{\mathsf{m}(4,1)\mathsf{m}(1,4)}_{\mathsf{lep}_3} & [s_2 & 4 & \{0,\ldots,4\} & Z()X(1)E(2,1)X(2)E(3,2)X(3)E(4,3) & Y()] \\
\hline
\xrightarrow{\mathsf{e}(4,3,4)}_{\mathsf{lep}_4} & [s_2 & 4 & \{0,\ldots,4\} & Z()X(1)E(2,1)X(2)E(3,2)X(3) & Y()A(3,4)] \\
\xrightarrow{\varepsilon}_{\mathsf{lep}_5} & [s_2 & 4 & \{0,\ldots,4\} & Z()X(1)E(2,1)X(2)E(3,2) & Y()A(3,4)] \\
\xrightarrow{\mathsf{e}(3,2,4)}_{\mathsf{lep}_4} & [s_2 & 4 & \{0,\ldots,4\} & Z()X(1)E(2,1)X(2) & Y()A(3,4)A(2,3)] \\
\xrightarrow{\varepsilon}_{\mathsf{lep}_6} & [s_2 & 2 & \{0,\ldots,4\} & Z()X(1)E(2,1) & Y()A(3,4)A(2,3)] \\
\xrightarrow{\mathsf{e}(2,1,2)}_{\mathsf{lep}_4} & [s_2 & 2 & \{0,\ldots,4\} & Z()X(1) & Y()A(3,4)A(2,3)A(1,2)] \\
\xrightarrow{\varepsilon}_{\mathsf{lep}_5} & [s_2 & 2 & \{0,\ldots,4\} & Z() & Y()A(3,4)A(2,3)A(1,2)] \\
\xrightarrow{\varepsilon}_{\mathsf{lep}_7} & [s_3 & 2 & \{0,\ldots,4\} & \varepsilon & Y()A(3,4)A(2,3)A(1,2)] \\
\hline
\xrightarrow{\mathsf{a}(1,2,2)}_{\mathsf{lep}_8} & [s_3 & 2 & \{0,\ldots,4\} & \varepsilon & Y()A(3,4)A(2,3)] \\
\xrightarrow{\mathsf{a}(2,3,2)}_{\mathsf{lep}_8} & [s_3 & 2 & \{0,\ldots,4\} & \varepsilon & Y()A(3,4)] \\
\xrightarrow{\mathsf{a}(3,4,2)}_{\mathsf{lep}_8} & [s_3 & 2 & \{0,\ldots,4\} & \varepsilon & Y()] \\
\xrightarrow{\varepsilon}_{\mathsf{lep}_9} & [s_4 & 2 & \{0,\ldots,4\} & \varepsilon & \varepsilon]
\end{array}
$$

Fig. 3. A run of the leader election protocol

3 Monadic Second-Order Logic

While DMPA serve as system models, we use monadic second-order logic to specify properties of data words. We assume countably infinite supplies of first-order and second-order variables. We let $x, y, \ldots$ denote first-order variables, which vary over word positions, and we use $X, Y, \ldots$ to denote second-order variables, which vary over sets of positions.

Definition 4 (MSO logic over data words). *The class* $\mathrm{MSO}_{\text{d-word}}(\Sigma, D)$ *of* monadic second-order (MSO) formulas *over data words is given by the following grammar, where* a *ranges over* Σ*, and* $1 \leq k, l \leq \max(\mathit{arity}(\Sigma))$*:*

$$\varphi ::= a(x) \mid d_k(x) = d_l(y) \mid x \leq y \mid x \in X \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x\varphi \mid \exists X\varphi$$

Formula $a(x)$ holds in a data word $w \in \Sigma_D^*$ if $\mathit{label}(i) = a$ when x is interpreted as $i \in \mathrm{dom}(w)$. Formula $d_k(x) = d_l(y)$ is satisfied wrt. interpretations i and j of x and y, respectively, if $k \leq \mathit{arity}(i)$, $l \leq \mathit{arity}(j)$, and $\mathit{data}_k(i) = \mathit{data}_l(j)$. Formula $x \leq y$, the boolean connectives, and quantifiers are self-explanatory. We also use the usual abbreviations $x < y$, $\forall x\varphi$, $\varphi \to \psi$ ...

For a data word w and a formula $\varphi(x_1, \ldots, x_n, X_1, \ldots, X_m)$ with free variables in $\{x_1, \ldots, x_n, X_1, \ldots, X_m\}$, we write $w, i_1, \ldots, i_n, I_1, \ldots, I_m \models \varphi$ if φ evaluates to true when interpreting the first-order variables by $i_1, \ldots, i_n \in \mathrm{dom}(w)$ and the second-order variables by $I_1, \ldots, I_m \subseteq \mathrm{dom}(w)$, respectively.

If φ is a sentence, i.e., it does not have any free variable, then we set $L(\varphi)$ to be the set of data words w such that $w \models \varphi$.

Example 5. We define a property satisfied by the 1-register 2-stack DMPA $\mathcal{A}_{lep}$ from Example 3 modeling the leader election protocol. To express that every new process will eventually receive an announcement containing a unique leader pid, we write $\varphi = \exists z \, \forall x \, (\mathsf{c}(x) \to \exists y \, (\mathsf{a}(y) \wedge x \leq y \wedge d_2(x) = d_2(y) \wedge d_1(z) = d_3(y)))$. We have $L(\mathcal{A}_{lep}) = L_2(\mathcal{A}_{lep}) \subseteq L(\varphi)$. □

Example 6. This example will show that MSO logic can be used to abstract from the linear order of a data word to model partially ordered behaviors.

We model concurrent programs where processes can fork other processes and exchange messages via send and receive primitives. Unlike in the leader election protocol, we will model sends and receives separately, which allows us a finer treatment when we formulate MSO properties. Again, processes have unique pids, which are modeled as data values. We let $D = \mathbb{N}$ be the pids and $\Sigma = \{\mathsf{s}, \mathsf{f}, !, ?\}$ be the labels. Label s takes one pid d, and $\mathsf{s}(d) \in \Sigma_D$ indicates that d has just started its execution. Labels f, !, and ? each take two arguments, one for the executing (i.e., forking, sending, receiving) process, and one for the communication partner (i.e., the new, receiving, sending process, respectively). In particular, $\mathsf{f}(c, d)$ is matched by $\mathsf{s}(d)$, and $!(c, d)$ is matched by $?(d, c)$. Figure 4 shows two data words, w_1 and w_2. The graphs above them illustrate their interpretation as the execution of a concurrent program, connecting a fork with a corresponding start action and a send with a matching receive. This connection will, in the following, be formalized in terms of MSO logic.

Rather than the linear order of the data word, we are interested in the causal dependencies in the underlying concurrent execution modeled by this data word, and these can be captured via MSO formulas. Let $x <_{\text{proc}} y$ be a shorthand for $x < y \wedge d_1(x) = d_1(y)$, which denotes that there is a process that executes first x and later y (we may say that a word position is "executed", as it is considered as a system event). In Figure 4, the relation induced by $x <_{\text{proc}} y$ is given in terms of the transitive closure of the horizontal edges. For example, $w_1, 1, 2 \models x <_{\text{proc}} y$ and $w_1, 1, 6 \models x <_{\text{proc}} y$. Now, consider formula $x <_{\text{m}} y$, which stands for $!(x) \wedge ?(y) \wedge x < y \wedge d_1(x) = d_2(y) \wedge d_2(x) = d_1(y)$. We assume a bound 1 on the channel capacities. To say that x and y form a message, we let $x <_{\text{msg}} y$ abbreviate $x <_{\text{m}} y \wedge \neg\exists z \, (x < z < y \wedge (x <_{\text{m}} z \vee z <_{\text{m}} y))$. For example, we have $w_1, 6, 7 \models x <_{\text{msg}} y$. Let us relate a fork position with the first position executed by the new process: $x <_{\text{fork}} y$ stands for $\mathsf{f}(x) \wedge \mathsf{s}(y) \wedge d_2(x) = d_1(y)$. For example, $w_1, 4, 5 \models x <_{\text{fork}} y$. To define causal dependencies between positions of a data word, we let $<_{\text{causal}}$ denote the transitive closure of the relation $<_{\text{proc}} \cup <_{\text{msg}} \cup <_{\text{fork}}$. It corresponds to the transitive closure of the edge relation depicted in Figure 4. Note that the transitive closure of an MSO definable binary relation is indeed MSO definable [7]. For example, $w_1, 1, 7 \models x <_{\text{causal}} y$, but $w_1, 4, 6 \not\models x <_{\text{causal}} y$ and $w_1, 6, 4 \not\models x <_{\text{causal}} y$.

Like in the leader election protocol, we assume a system architecture that allows us to pass pids along messages or process forks. When we consider concrete implementations of such concurrent programs, it is crucial that a process sending a message to another knows the pid of the receiving process. We will determine an MSO formula $\varphi_{realizable}$ that checks a system for such consistency (*realizability* in

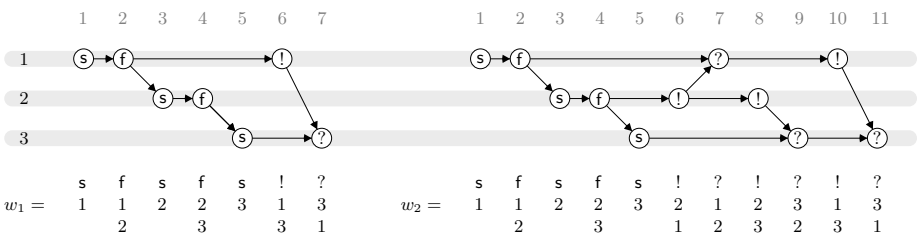

Fig. 4. Two data words

the terminology of [5]). It uses a formula $knows(x, y)$, which holds if the process executing x, right before performing the action, knows the pid of the process executing y. Regarding Figure 4, we would like to have $w_1, 6, 7 \not\models knows(x, y)$, as there is no way to communicate pid 3 to the process with pid 1. On the other hand, we will have $w_2, 10, 11 \models knows(x, y)$ as pid 3 can be communicated to process 1 along the message from 2 (which spawned 3) to 1. We first describe a formula $x <_{\text{flow}} y$, which intuitively says that there is some flow of information from position x to position y. In other words, pids can be passed from the process executing x to the process executing y. We set $<_{\text{flow}}$ to be the transitive closure of $<_{\text{causal}} \cup <_{\text{fork}}^{-1}$. For example, $w_1, 3, 6 \models x <_{\text{flow}} y$, but $w_1, 4, 6 \not\models x <_{\text{flow}} y$. Now, we set $knows(x, y)$ to be $\exists y' \, (y' <_{\text{proc}} y \wedge flow(y', x))$. Finally, we consider our system to be realizable if it satisfies $\varphi_{realizable} := \forall x \forall y \, (x <_{\text{msg}} y \rightarrow knows(x, y))$. We have $w_1 \not\models \varphi_{realizable}$ but $w_2 \models \varphi_{realizable}$.

In a communicating system, one would like to avoid *races*. A race occurs when two receives of the same process are ordered in some way while their send events (from different processes) are independent. This may be seen as a design error. Formally, a data word has a race if it satisfies $\varphi_{race} := \exists x, y, x', y' \, (x <_{\text{msg}} y \wedge x' <_{\text{msg}} y' \wedge y <_{\text{proc}} y' \wedge \neg(x <_{\text{causal}} x'))$. We have $w_1 \not\models \varphi_{race}$ but $w_2 \models \varphi_{race}$. □

4 Model Checking DMPA

We now present our main result, decidability of the model-checking problem for (bounded control change) DMPA wrt. MSO logic:

Theorem 7. *For a DMPA $\mathcal{A}$ over (Σ, D), a natural number $\ell \geq 1$, and a sentence $\varphi \in \text{MSO}_{\text{d-word}}(\Sigma, D)$, one can decide if $L_\ell(\mathcal{A}) \subseteq L(\varphi)$.*

The rest of this section is devoted to the proof of Theorem 7, which we outline in the following. First, we represent a run ρ of a DMPA as a *(multiply) nested data word*. The nested data word associated with ρ is the concatenation of the instantiations of transitions used in ρ. In addition, it has nesting edges from a pushed stack symbol to the position where it is popped. There is a precise correspondence between (ℓ-phase) runs and (certain ℓ-phase) nested data words.

Moreover, the data word generated by ρ will be exactly the word projection (without nesting edges) of its nested word onto the alphabet Σ_D. Next, we look at *abstract* nested words, which, instead of data values, contain the parameters used in the run. We, therefore, deal with nested words over a finite alphabet. A nested data word and the abstract version corresponding to a run are depicted in Figure 5. The trick is now that we can, using the nesting edges, define MSO formulas over abstract nested words that recover equality of data values in the concrete version. As the set of abstract nested words that correspond to accepting runs of the DMPA is also definable in MSO logic, we reduce, in this way, the model-checking problem for a DMPA to a satisfiability problem over abstract nested words. Satisfiability of MSO formulas over ℓ-phase nested words is decidable due to [15] so that the theorem follows.

Nested Words. Let $\mathbb{h} \geq 1$. An $\mathbb{h}$-*stack alphabet* is a (possibly infinite) alphabet Γ together with mappings $\mathit{stack} : \Gamma \to \{0, 1, \ldots, \mathbb{h}\}$ and $\mathit{type} : \Gamma \to \{\mathsf{push}, \mathsf{pop}, \mathsf{int}\}$ such that, for all $a \in \Gamma$, we have $\mathit{type}(a) = \mathsf{int}$ iff $\mathit{stack}(a) = 0$. Given $w = a_1 \ldots a_n \in \Gamma^*$ and $i \in \mathrm{dom}(w)$, we let $\mathit{stack}(i) = \mathit{stack}(a_i)$ and $\mathit{type}(i) = \mathit{type}(a_i)$. For $t \in [\mathbb{h}]$, we call $w \in \Gamma^*$ *t-well-nested* if it can be generated by the context-free grammar $A ::= aAb \mid AA \mid \varepsilon \mid c$ where $a, b, c \in \Gamma$ are such that $\mathit{stack}(a) = \mathit{stack}(b) = t \neq \mathit{stack}(c)$, $\mathit{type}(a) = \mathsf{push}$, and $\mathit{type}(b) = \mathsf{pop}$.

A nested word over Γ is a pair $W = (w, \curvearrowright)$ where $w \in \Gamma^*$ and $\curvearrowright \subseteq \mathrm{dom}(w) \times \mathrm{dom}(w)$ is the binary *matching relation*, which is uniquely determined as follows: for all $i, j \in \mathrm{dom}(w)$, $i \curvearrowright j$ iff $i < j$ and there is $t \in [\mathbb{h}]$ such that $\mathit{stack}(i) = \mathit{stack}(j) = t$, $\mathit{type}(i) = \mathsf{push}$, $\mathit{type}(j) = \mathsf{pop}$, and $a_{i+1} \ldots a_{j-1}$ is t-well-nested. Note that there might be push or pop positions that are not matched wrt. $\curvearrowright$. The set of nested words over Γ is denoted by $\mathit{Nested}(\Gamma)$.

Let $\ell \geq 1$. A nested word $(w, \curvearrowright)$ is an ℓ-*phase nested word* if w can be written as $w_1 \ldots w_\ell$ with $w_i \in \Gamma^*$ where, for all $i \in \{1, \ldots, \ell\}$, there is $t \in [\mathbb{h}]$ such that, for each letter $a \in \Gamma$ that occurs in w_i, $\mathit{type}(a) = \mathsf{pop}$ implies $\mathit{stack}(a) = t$.

The class $\mathrm{MSO}_{\mathrm{nw}}(\Gamma)$ of MSO formulas over nested words is given by the following grammar, where a ranges over Γ:

$$\varphi ::= a(x) \mid x \curvearrowright y \mid x \leq y \mid x \in X \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x\varphi \mid \exists X\varphi$$

The atomic predicates are interpreted over a nested word $W = (a_1 \ldots a_n, \curvearrowright)$ as follows: $W, i \models a(x)$ if $a_i = a$, $W, i, j \models x \curvearrowright y$ if $i \curvearrowright j$, and $W, i, j \models x \leq y$ if $i \leq j$. The other connectives are as expected.

Theorem 8 (La Torre et al. [15]). *Given a finite $\mathbb{h}$-stack alphabet Γ, $\ell \geq 1$, and a sentence $\varphi \in \mathrm{MSO}_{\mathrm{nw}}(\Gamma)$, one can decide if there is an ℓ-phase nested word W over Γ such that $W \models \varphi$.*

Nested Data Words. Next, we define nested words carrying data values. Let Γ be a finite *ranked* $\mathbb{h}$-stack alphabet, i.e., every letter $a \in \Gamma$ has some $\mathit{arity}_\Gamma(a) \in \mathbb{N}$. We can interpret Γ_D as an infinite $\mathbb{h}$-stack alphabet in the obvious manner. A *nested data word* over (Γ, D) is a nested word over Γ_D. Notions

from data words such as $\mathrm{dom}(w)$ and $data_k(i)$ can be transferred to nested data words $W = (w, \curvearrowright)$ by applying them to the w-component.

The set $\mathrm{MSO}_{\text{d-nw}}(\Gamma, D)$ of MSO formulas over nested data words is given by the following grammar, where a ranges over Γ, and $1 \leq k, l \leq \max(arity(\Gamma))$:

$$\varphi ::= a(x) \mid d_k(x) = d_l(y) \mid x \curvearrowright y \mid x \leq y \mid x \in X \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x\varphi \mid \exists X\varphi$$

We omit the definition of the semantics, which is as expected.

Suppose $\Sigma \subseteq \Gamma$. Given a nested data word W over (Γ, D), we denote by $\mathsf{Proj}_\Sigma(W)$ the data word from Σ_D^* obtained by restricting (or projecting) W to Σ_D and discarding $\curvearrowright$. Using a simple relativization, we obtain:

Proposition 9. *Let $\Sigma \subseteq \Gamma$ and $\varphi \in \mathrm{MSO}_{\text{d-word}}(\Sigma, D)$ be a sentence. We can effectively construct a sentence $\widetilde{\varphi} \in \mathrm{MSO}_{\text{d-nw}}(\Gamma, D)$ such that, for all $W \in Nested(\Gamma_D)$, we have $W \models \widetilde{\varphi}$ iff $\mathsf{Proj}_\Sigma(W) \models \varphi$.*

Parse Words. Let $\ell \geq 1$ and $\mathcal{A} = (S, \mathcal{Z}, s_0, Z, F, \Delta)$ be a ($\mathbb{k}$-register, $\mathbb{h}$-stack) DMPA over (Σ, D). Without loss of generality, we assume that there is a mapping $stack : \mathcal{Z} \to [\mathbb{h}]$ such that each stack symbol $A \in \mathcal{Z}$ is written on/removed from $stack(A)$ only. We assume $stack(Z) = 1$. We define a finite ranked $\mathbb{h}$-stack alphabet $\Gamma = \Sigma \uplus \mathcal{Z} \uplus \overline{\mathcal{Z}} \uplus S \uplus \overline{S}$ where $\overline{\mathcal{Z}} = \{\overline{A} \mid A \in \mathcal{Z}\}$ and $\overline{S} = \{\overline{s} \mid s \in S\}$ contain a marked copy of every letter from $\mathcal{Z}$ and S, respectively. We retain the arities defined by the alphabets Σ and $\mathcal{Z}$. We let $arity_\Gamma(\overline{s}) = arity_\Gamma(s) = \mathbb{k}$ for all $s \in S$ and $arity_\Gamma(\overline{A}) = arity_\Gamma(A)$ for all $A \in \mathcal{Z}$. Moreover, $type(a) = \mathsf{int}$ and $stack(a) = 0$ for all $a \in \Sigma \cup S \cup \overline{S}$. Finally, $stack(\overline{A}) = stack(A)$, $type(A) = \mathsf{push}$, and $type(\overline{A}) = \mathsf{pop}$ for all $A \in \mathcal{Z}$.

Let $\mathcal{O} \subset \Pi$ denote the finite set of parameters occurring in $\mathcal{A}$. Recall that, by $\Gamma_\mathcal{O}$, we denote the (finite) $\mathbb{h}$-stack alphabet $\{a(\pi_1, \ldots, \pi_m) \mid a \in \Gamma,\ m = arity_\Gamma(a), \text{ and } \pi_1, \ldots, \pi_m \in \mathcal{O}\}$. For $b = a(\pi_1, \ldots, \pi_m) \in \Gamma_\mathcal{O}$ and $1 \leq k \leq m$, we denote by $\mathsf{par}_k(b)$, its k-th parameter π_k.

We are now ready to define the (abstract and concrete) parse words of $\mathcal{A}$. Consider a transition $\delta = t{:}A, s \xrightarrow{\Phi, u, \mathsf{upd}} s'$ with $\mathsf{upd} = (\pi_1, \ldots, \pi_\mathbb{k}, u_1, \ldots, u_\mathbb{h})$. Note that $t = stack(A)$ by assumption. Let $m = arity(A)$. We define the string of δ as $string(\delta) := \overline{s}(r_1, \ldots, r_\mathbb{k})\overline{A}(p_1, \ldots, p_m)uu_1 \ldots u_\mathbb{h}s'(\pi_1, \ldots, \pi_\mathbb{k}) \in \Gamma_\mathcal{O}^*$. For instance, for the DMPA $\mathcal{A}_{lep}$ from Example 3, we have

$$string(\mathsf{lep}_1) = \overline{s_0}(r_1)\overline{Z}()Z()C(q_1)Y()s_1(q_1)$$
$$string(\mathsf{lep}_2) = \overline{s_1}(r_1)\overline{C}(p_1)\mathsf{c}(p_1, q_1)X(p_1)E(q_1, p_1)C(q_1)s_1(r_1)$$

For an interpretation $\sigma : \mathcal{O} \to D$, we define similarly the string of the concrete transition $\sigma(\delta)$ with data values $\sigma(\pi)$ substituted for parameters $\pi \in \mathcal{O}$. It is denoted $string(\sigma(\delta))$. Note that the string of a transition does not consider guards. Guards are taken into account later, in Proposition 14.

Consider a run ρ of $\mathcal{A}$ of the form

$$\gamma_0 \xRightarrow{w_1}_{\sigma_1, \delta_1} \gamma_1 \xRightarrow{w_2}_{\sigma_2, \delta_2} \cdots \xRightarrow{w_n}_{\sigma_n, \delta_n} \gamma_n$$

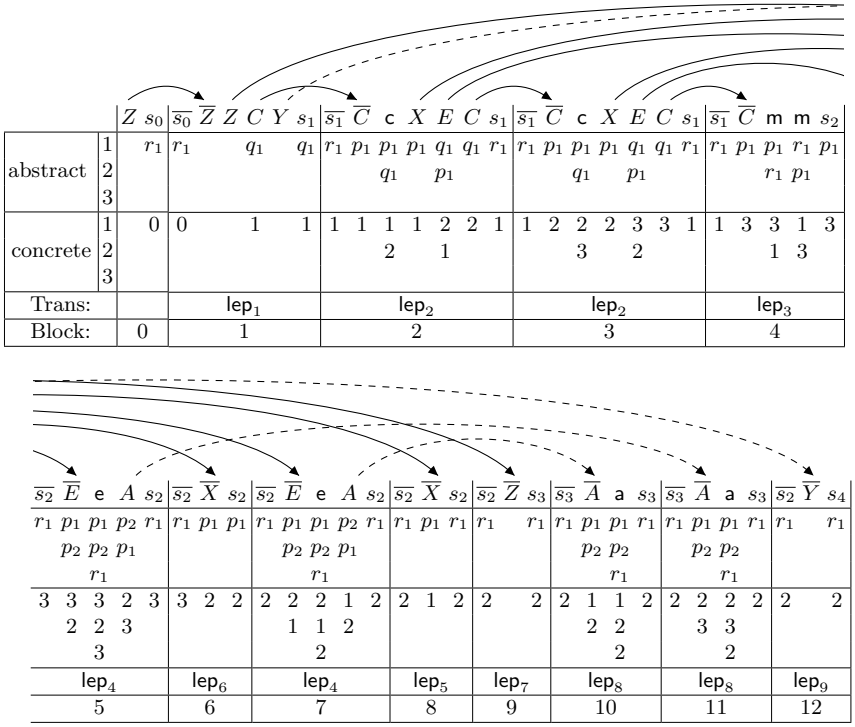

Fig. 5. An abstract and a concrete parse word (split over two lines)

The nested word $(w, \curvearrowright)$ with $w = Z()s_0(r_1, \ldots, r_{\mathbb{k}})string(\delta_1) \ldots string(\delta_n) \in \Gamma_{\mathcal{O}}^*$ is the *abstract parse word* of ρ and denoted apw_ρ. The nested data word $(w', \curvearrowright)$ with $w' = Z()s_0(\sigma_1(r_1), \ldots, \sigma_1(r_{\mathbb{k}}))string(\sigma_1(\delta_1)) \ldots string(\sigma_n(\delta_n)) \in \Gamma_D^*$ is the *concrete parse word* of ρ denoted pw_ρ. Notice that $\mathrm{dom}(\mathsf{pw}_\rho) = \mathrm{dom}(\mathsf{apw}_\rho)$. Moreover, ρ is ℓ-phase iff pw_ρ is ℓ-phase iff apw_ρ is ℓ-phase.

Figure 5 illustrates an abstract parse word and a concrete parse word of the run $\mathsf{lep}_1\mathsf{lep}_2\mathsf{lep}_2\mathsf{lep}_3\mathsf{lep}_4\mathsf{lep}_6\mathsf{lep}_4\mathsf{lep}_5\mathsf{lep}_7\mathsf{lep}_8\mathsf{lep}_8\mathsf{lep}_9$ of DMPA $\mathcal{A}_{lep}$ from Example 3. The curved lines depict the nesting relation $\curvearrowright$ (straight lines for stack 1, dotted lines for stack 2).

The data word $\sigma(u)$ generated by a concrete transition $\sigma(\delta)$ is precisely the Σ-projection of $string(\sigma(\delta))$. Hence, the data word generated by a run ρ is $\mathsf{Proj}_\Sigma(\mathsf{pw}_\rho)$. The data words accepted by $\mathcal{A}$ with ℓ-phase runs are the Σ-projections of the concrete parse words of these runs:

Proposition 10. *We have* $L_\ell(\mathcal{A}) = \{\mathsf{Proj}_\Sigma(\mathsf{pw}_\rho) \in \Sigma_D^* \mid \rho$ *is an* ℓ*-phase accepting run of* $\mathcal{A}\}$.

An abstract parse word is an abstraction of several concrete parse words. Our aim is to recover from an abstract parse word all data equalities that hold in the concrete parse word. To do so, we will define formulas $\varphi_{k,l}(x, y) \in \mathrm{MSO}_{\mathrm{nw}}(\Gamma_{\mathcal{O}})$

for all $1 \leq k, l \leq \max(\mathit{arity}(\Gamma))$, with free variables x and y. Intuitively, $\varphi_{k,l}(x, y)$ will hold in an abstract parse word iff $d_k(x) = d_l(y)$ holds in any corresponding concrete parse word.

We first give some definitions and macros. Block 0 of an abstract parse word consists of the first two positions, which are labelled $Z()$ and $s_0(r_1, \ldots, r_{\mathbb{k}})$ respectively. Then, we find a concatenation of blocks of the form $\mathit{string}(\delta) = \overline{s}(r_1, \ldots, r_{\mathbb{k}})\overline{A}(p_1, \ldots, p_m)vs'(\pi_1, \ldots, \pi_{\mathbb{k}})$ for some transition δ (see Figure 5). For any position x, we denote by $\mathsf{Block}(x)$ the block of x. We use $\mathsf{Block}(x) = \mathsf{Block}(y)$ to state that x and y belong to the same block, which can be expressed by the following first-order formula: $x, y \leq 2 \vee \exists x', y' \, (x' \leq x, y \leq y' \wedge \bigvee_{\overline{s} \in \overline{S}_{\mathcal{O}}} \overline{s}(x') \wedge \bigvee_{s \in S_{\mathcal{O}}} s(y') \wedge \forall z \, ((z < y' \wedge \bigvee_{\overline{s} \in \overline{S}_{\mathcal{O}}} \overline{s}(z)) \rightarrow z \leq x'))$. Moreover, we will use the macro $\mathsf{Block}(x) \leq \mathsf{Block}(y) := x \leq y \vee \mathsf{Block}(x) = \mathsf{Block}(y)$.

Let $\mathsf{par}_k(x)$ denote the k-th parameter of position x of the abstract parse word. The macro $\mathsf{par}_k(x) = \mathsf{par}_l(y)$ says that x and y carry the same parameter at indices k and l, respectively. It is the disjunction of formulas $b(x) \wedge b'(y)$ where $b, b' \in \Gamma_{\mathcal{O}}$ are such that $\mathsf{par}_k(b) = \mathsf{par}_l(b')$. We also let $\mathsf{existspar}_k(x)$ be the disjunction of formulas $b(x)$ where $b \in \Gamma_{\mathcal{O}}$ is such that $\mathit{arity}(b) \geq k$.

Let us see how to propagate a data value to a *later block*. Clearly, we have $d_k(x) = d_l(y)$ in the concrete parse word if in the abstract parse word the formula

$$\psi_{k,l}(x, y) := \begin{pmatrix} \mathsf{Block}(x) = \mathsf{Block}(y) \wedge \mathsf{par}_k(x) = \mathsf{par}_l(y) \\ \vee \, \mathsf{Block}(x) \neq \mathsf{Block}(y) \wedge x + 1 = y \wedge k = l \leq \mathbb{k} \\ \vee \, x \curvearrowright y \wedge \bigvee_{a = A(\ldots) \in \mathcal{Z}_{\mathcal{O}}} (a(x) \wedge k = l \leq \mathit{arity}(A)) \end{pmatrix}$$

holds. Note that $\psi_{k,l}(x, y)$ implies $\mathsf{Block}(x) \leq \mathsf{Block}(y)$ and that $\psi_{k,k}(x, x)$ is equivalent to $\mathsf{existspar}_k(x)$. Data equality is also ensured if we can reach y from x using a sequence of $\psi_{i,j}$ steps. It is well-known that such a "transitive closure" can be defined in MSO: we let $m = \max(\mathit{arity}(\Gamma))$ and define $x \; {}_k\!\rightsquigarrow_l \; y$ by

$$\mathsf{existspar}_k(x) \wedge \forall X_1, \ldots, X_m \\ \Big(x \in X_k \wedge \bigwedge_{1 \leq i,j \leq m} \forall z_1, z_2 \, (z_1 \in X_i \wedge \psi_{i,j}(z_1, z_2)) \rightarrow z_2 \in X_j \Big) \rightarrow y \in X_l$$

so that we have $\mathsf{apw}_\rho, i, j \models x \; {}_k\!\rightsquigarrow_l \; y$ iff for some $n > 0$, there are sequences $i = i_0, i_1, \ldots, i_n = j$ and $k = k_0, k_1, \ldots, k_n = l$ such that $\mathsf{apw}_\rho, i_p, i_{p+1} \models \psi_{k_p, k_{p+1}}(x, y)$ for all $0 \leq p < n$. Therefore, $x \; {}_k\!\rightsquigarrow_l \; y$ in the abstract parse word implies $d_k(x) = d_l(y)$ in the concrete parse word.

For the general case, we will prove that $d_k(x) = d_l(y)$ in the concrete parse word iff there exists a position z such that the i-th value of z was propagated to x as its k-th value and to y as its l-th value. So we define

$$\varphi_{k,l}(x, y) := \exists z \bigvee_i \; z \; {}_i\!\rightsquigarrow_k \; x \wedge z \; {}_i\!\rightsquigarrow_l \; y$$

Example 11. We will see how the formulas defined above retrieve data equality on the abstract parse word from Figure 5. Let us check whether the data at the third component of e in Block 5 is same as the data at the second component of

a in Block 11, i.e., whether r_1 in Block 5 and p_2 of Block 11 hold the same value. First, p_2 in Block 11 equals p_1 in Block 5, which equals q_1 in Block 3, which is fresh. Similarly, r_1 in Block 5 is the same as p_1 in Block 4, which equals q_1 in Block 3, which is fresh. Hence, from q_1 of Block 3, we can reach both r_1 in Block 5 and p_2 in Block 11 by $\rightsquigarrow$, thus concluding they hold the same data value. □

We can prove that the formulas $\varphi_{k,l}(x,y)$ defined above are indeed correct.

Proposition 12. *For all runs ρ of $\mathcal{A}$ and all positions $i,j \in \mathrm{dom}(\mathsf{pw}_\rho)$, we have $\mathsf{pw}_\rho, i, j \models d_k(x) = d_l(y)$ iff $\mathsf{apw}_\rho, i, j \models \varphi_{k,l}(x,y)$.*

Proof. Fix a run ρ and let $i,j \in \mathrm{dom}(\mathsf{pw}_\rho)$.

First, we assume that $\mathsf{pw}_\rho, i, j \models d_k(x) = d_l(y)$, i.e., $data_k(i) = data_l(j)$. We show $\mathsf{apw}_\rho, i, j \models \varphi_{k,l}(x,y)$. The proof is by induction on the sum $\mathsf{Block}(i) + \mathsf{Block}(j)$. If $\mathsf{Block}(i) = \mathsf{Block}(j) = 0$ then we must have $i = j = 2$ and $k = l \leq \mathbb{k}$. We deduce that $\mathsf{apw}_\rho, i, j \models \psi_{k,l}(x,y)$ and we are done.

Assume now that $\mathsf{Block}(i) + \mathsf{Block}(j) > 0$. Without loss of generality, we assume that $\mathsf{Block}(i) \leq \mathsf{Block}(j)$. There are three cases to consider.

1. Suppose the l-th parameter of position j is some register r_n with $n \leq \mathbb{k}$. Let $j' \leq j$ be the first position of $\mathsf{Block}(j)$. By definition, the n-th parameter of j' is r_n and $\mathsf{apw}_\rho, j', j \models \psi_{n,l}(x,y)$. Next, $j'-1$ is the last position of $\mathsf{Block}(j)-1$ and we have $\mathsf{apw}_\rho, j'-1, j' \models \psi_{n,n}(x,y)$. We deduce that $data_n(j'-1) = data_n(j') = data_l(j)$, hence also $data_k(i) = data_n(j'-1)$. By induction, we obtain $\mathsf{apw}_\rho, i, j'-1 \models \varphi_{k,n}(x,y)$, which implies $\mathsf{apw}_\rho, i, j \models \varphi_{k,l}(x,y)$.
2. Suppose the l-th parameter of position j is some $p_n \in P$. Let $j' \leq j$ be the second position of $\mathsf{Block}(j)$. By definition, the n-th parameter of j' is p_n and $\mathsf{apw}_\rho, j', j \models \psi_{n,l}(x,y)$. Next, let j'' be such that $j'' \curvearrowright j'$ so that we have $\mathsf{apw}_\rho, j'', j' \models \psi_{n,n}(x,y)$. We deduce that $data_n(j'') = data_n(j') = data_l(j)$, hence also $data_k(i) = data_n(j'')$. By induction, we obtain $\mathsf{apw}_\rho, i, j'' \models \varphi_{k,n}(x,y)$, which implies $\mathsf{apw}_\rho, i, j \models \varphi_{k,l}(x,y)$.
3. Suppose the l-th parameter of position j is some $q \in Q$. Then, $data_l(j)$ is fresh and never occured in a previous block. Since $data_k(i) = data_l(j)$, we deduce that $\mathsf{Block}(i) = \mathsf{Block}(j)$. Moreover, the k-th parameter at position i cannot be in $R \cup P$ since otherwise, by the previous cases, the value $data_k(i)$ would have occurred in a previous block. Since different parameters from Q always get distinct values, we deduce that the k-th parameter at position i must be q, which implies $\mathsf{apw}_\rho, i, j \models \varphi_{k,l}(x,y)$.

Conversely, we assume that $\mathsf{apw}_\rho, i, j \models \varphi_{k,l}(x,y)$. We show that $data_k(i) = data_l(j)$. Since data equality is transitive, we may assume without loss of generality that $\mathsf{apw}_\rho, i, j \models x \ {}_k{\rightsquigarrow}_l \ y$. Hence, we find $n > 0$ and two sequences $i = i_0, i_1, \ldots, i_n = j$ and $k = k_0, k_1, \ldots, k_n = l$ such that for all $0 \leq p < n$ we have $\mathsf{apw}_\rho, i_p, i_{p+1} \models \psi_{k_p,k_{p+1}}(x,y)$. We deduce $data_{k_p}(i_p) = data_{k_{p+1}}(i_{p+1})$ for all $0 \leq p < n$ and we are done. □

Corollary 13. *For every sentence $\xi \in \mathrm{MSO}_{\text{d-nw}}(\Gamma, D)$, we can effectively construct a sentence $\widehat{\xi} \in \mathrm{MSO}_{\mathrm{nw}}(\Gamma_{\mathcal{O}})$ such that, for all runs ρ of $\mathcal{A}$, we have $\mathsf{pw}_\rho \models \xi$ iff $\mathsf{apw}_\rho \models \widehat{\xi}$.*

Proof. We obtain $\widehat{\xi}$ by replacing every occurrence of $d_k(x) = d_l(y)$ in ξ with $\varphi_{k,l}(x, y)$, and every occurrence of $a(x)$ with the disjunction of formulas $b(x)$ where $b = a(\pi_1, \ldots, \pi_m) \in \Gamma_{\mathcal{O}}$. The result then follows from Proposition 12. □

The last proposition needed for our proof says that the set of abstract parse words of ℓ-phase accepting runs of $\mathcal{A}$ is MSO definable.

Proposition 14. *There is $\psi \in \mathrm{MSO}_{\mathrm{nw}}(\Gamma_{\mathcal{O}})$ such that $L(\psi) = \{\mathsf{apw}_\rho \mid \rho$ is an ℓ-phase accepting run of $\mathcal{A}\}$.*

Proof. First observe that, given $\mathcal{A}$, the set $\mathrm{Str}_{\mathcal{A}} = \{\mathit{string}(\delta) \mid \delta$ is a transition of $\mathcal{A}\}$ is finite. The formula $\psi \in \mathrm{MSO}_{\mathrm{nw}}(\Gamma_{\mathcal{O}})$ is the conjunction of formulas saying the following:

- The word projection of a nested word is in $Z()s_0(r_1, \ldots, r_{\mathbb{k}}) \cdot \{\mathrm{Str}_{\mathcal{A}}\}^*$.
- The k-phase restriction is respected.
- The transitions should respect the state. This can be ensured by saying that if the parse nested word admits a factor $s_1(\pi_1, \ldots, \pi_{\mathbb{k}})\overline{s_2}(r_1, \ldots, r_{\mathbb{k}})$ with $s_1, s_2 \in S$, then $s_1 = s_2$.
- There is an assignment of blocks to guards that comply with the transition relation. The guards have to be satisfied, which is checked using the formulas $\varphi_{k,l}(x, y)$.
- A transition of the form $t{:}A, s \xrightarrow{\Phi,u,\mathsf{upd}} s'$ can be fired only if the top symbol of t has label A. To take care of this, we say that every pop position with label $\overline{A}$ has a $\curvearrowright$-matched push position with label A.
- The last position is labelled with an accepting state.

With this, we have $L(\psi) = \{\mathsf{apw}_\rho \mid \rho$ is an ℓ-phase accepting run of $\mathcal{A}\}$ as required. □

We are now ready to prove our main result:

Proof (of Theorem 7). Let $\varphi \in \mathrm{MSO}_{\text{d-word}}(\Sigma, D)$. We consider the formula $\widehat{\widetilde{\varphi}} \in \mathrm{MSO}_{\mathrm{nw}}(\Gamma_{\mathcal{O}})$ obtained from Proposition 9 and Corollary 13. We show that $L_\ell(\mathcal{A}) \subseteq L(\varphi)$ iff the formula $\psi \to \widehat{\widetilde{\varphi}}$ is valid over $\mathit{Nested}(\Gamma_{\mathcal{O}})$ where ψ is from Proposition 14. This validity is decidable due to Theorem 8.
$\Longrightarrow$: Assume $L_\ell(\mathcal{A}) \subseteq L(\varphi)$. Let $W \in \mathit{Nested}(\Gamma_{\mathcal{O}})$ be such that $W \models \psi$. By Proposition 14, there is an ℓ-phase accepting run ρ of $\mathcal{A}$ such that $W = \mathsf{apw}_\rho$. By Proposition 10, we get $\mathsf{Proj}_\Sigma(\mathsf{pw}_\rho) \in L_\ell(\mathcal{A})$. Hence $\mathsf{Proj}_\Sigma(\mathsf{pw}_\rho) \models \varphi$ and, by Proposition 9, we get $\mathsf{pw}_\rho \models \widetilde{\varphi}$. Finally, Corollary 13 implies $W = \mathsf{apw}_\rho \models \widehat{\widetilde{\varphi}}$.
$\Longleftarrow$: Assume that $\psi \to \widehat{\widetilde{\varphi}}$ is valid. Let $w \in L_\ell(\mathcal{A})$. By Proposition 10, there is an ℓ-phase accepting run ρ of $\mathcal{A}$ such that $w = \mathsf{Proj}_\Sigma(\mathsf{pw}_\rho)$. By Proposition 14, we have $\mathsf{apw}_\rho \models \psi$ and since $\psi \to \widehat{\widetilde{\varphi}}$ is valid we get $\mathsf{apw}_\rho \models \widehat{\widetilde{\varphi}}$. We obtain $\mathsf{pw}_\rho \models \widetilde{\varphi}$ by Corollary 13 and, finally, $w = \mathsf{Proj}_\Sigma(\mathsf{pw}_\rho) \models \varphi$ by Proposition 9. □

5 Conclusion

In this paper, we introduced DMPA and showed that their model-checking problem is decidable wrt. the full MSO logic over data words. Note that this contributes to the area of parametrized verification [1], as model checking can prove that a property holds for any number of processes.

An important next step is to bridge the gap between DMPA specifications and concrete implementations, for example in terms of automata with pid-passing capabilities [4, 5]. Recall that the leader election protocol requires a context-sensitive specification when we define its behavior globally. However, it can be implemented as a finite-state system when we assume several local copies of processes that can send and receive messages as well as process identities. It remains to identify classes of DMPA that can be implemented in this way.

Recall that our main result relies on Theorem 8, whose proof [15, 18] essentially shows that ℓ-phase nested words have bounded tree width. It would be worthwhile to study if one can reduce our model-checking problem to a satisfiability problem for MSO logic over some class of graphs of bounded tree width or bounded clique width.

References

1. P. A. Abdulla. Forcing monotonicity in parameterized verification: From multisets to words. In J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, and B. Rumpe, editors, *SOFSEM'10*, volume 5901 of *LNCS*, pages 1–15. Springer, 2010.
2. M. Bojańczyk, C. David, A. Muscholl, Th. Schwentick, and L. Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27, 2011.
3. M. Bojańczyk, A. Muscholl, Th. Schwentick, and L. Segoufin. Two-variable logic on data trees and applications to XML reasoning. *J. ACM*, 56(3), 2009.
4. B. Bollig. An automaton over data words that captures EMSO logic. In J.-P. Katoen and B. König, editors, *CONCUR'11*, volume 6901 of *LNCS*, pages 171–186. Springer, 2011.
5. B. Bollig and L. Hélouët. Realizability of dynamic MSC languages. In F. M. Ablayev and E. W. Mayr, editors, *CSR'10*, volume 6072 of *LNCS*, pages 48–59, 2010.
6. E. Y. C. Cheng and M. Kaminski. Context-free languages over infinite alphabets. *Acta Inf.*, 35(3):245–267, 1998.
7. B. Courcelle. Graph rewriting: an algebraic and logic approach. In *Handbook of theoretical computer science (vol. B)*, pages 193–242. MIT Press, 1990.
8. C. David, L. Libkin, and T. Tan. On the satisfiability of two-variable logic over data words. In C. G. Fermüller and A. Voronkov, editors, *LPAR 2010*, LNCS, pages 248–262. Springer, 2010.
9. S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic*, 10(3), 2009.
10. S. Demri, R. Lazić, and A. Sangnier. Model checking freeze LTL over one-counter automata. In R. M. Amadio, editor, *FoSSaCS'08*, volume 4962 of *LNCS*, pages 490–504. Springer, 2008.

11. S. Demri and A. Sangnier. When model-checking freeze LTL over counter machines becomes decidable. In C.-H. L. Ong, editor, *FoSSaCS'10*, volume 6014 of *LNCS*. Springer, 2010.
12. O. Grumberg, O. Kupferman, and S. Sheinvald. Variable automata over infinite alphabets. In A. H. Dediu, H. Fernau, and C. Martín-Vide, editors, *LATA'10*, volume 6031 of *LNCS*, pages 561–572. Springer, 2010.
13. M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.
14. M. Kaminski and D. Zeitlin. Finite-memory automata with non-deterministic reassignment. *Int. J. Found. Comput. Sci.*, 21(5):741–760, 2010.
15. S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS'07*, pages 161–170. IEEE Computer Society Press, 2007.
16. S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS'08*, LNCS, pages 299–314. Springer, 2008.
17. M. Leucker, P. Madhusudan, and S. Mukhopadhyay. Dynamic message sequence charts. In *FSTTCS'02*, volume 2556 of *LNCS*, pages 253–264. Springer, 2002.
18. P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In Th. Ball and M. Sagiv, editors, *POPL '11*, pages 283–294. ACM, 2011.
19. M. Niewerth and T. Schwentick. Two-variable logic and key constraints on data words. In T. Milo, editor, *ICDT'11*, pages 138–149. ACM, 2011.
20. N. Tzevelekos. Fresh-register automata. In Th. Ball and M. Sagiv, editors, *POPL'11*, pages 295–306. ACM, 2011.